

TCP protokol

Za razliku od UDP protokola, TCP je protocol koji se zasniva prvenstveno na kreiranju konekcije između klijenta i servera i tek nakon njenog ostvarivanja sledi slanje podataka. Ovakav tip komunikacije se realizuje pomoću procedure trostrukog usaglašavanja (three-way handshake). Kada želimo da kreiramo TCP konekciju, dodeljujemo joj soket. Pošto ostvarimo konekciju kroz soket, kada jedna strana (bilo klijent bilo server) želi da pošalje podatak preko prethodno ostvarene konekcije – taj podatak smešta u svoj soket.

Procedura trostrukog usaglašavanja (three-way handshake)

Kako bi mogao da odgovori na zahtev klijenta, server mora biti spreman, što podrazumeva dve stvari:

- kao i UDP, i TCP server mora prethodno biti pokrenut;
- server mora imati poseban soket za prihvatanje inicijalnog zahteva za konekciju od strane klijenta.

Pošto imamo pokrenut server, klijent može inicirati TCP konekciju pomoću TCP soketa. TCP klijent kreira soket u kojem zadaje adresu serverskog soketa.

U toku trostrukog usaglašavanja, klijent govori serveru da želi konekciju. Nakon što server registruje taj zahtev, otvara novi, poseban, soket koji je namenjen samo tom klijentu i komunikaciji sa njim i šalje potvrdu prijema. Nakon prispeća te poruke klijent je uspešno povezan na soket, komunikacija je omogućena i obe strane su spremne za slanje.

Kao i u primeru sa UDP komunikacijom, i u TCP komunikaciji ćemo realizovati klijent-server (tcp_local_client.py i tcp_local_server.py) strukturu koja će na serverskoj strani dobiti poruku podeliti na osnovu karaktera zapeta (,) i klijentu vratiti broj elemenata dobijenih nakon te podele.

TCP klijent

Kada se pogleda prvih par linija TCP klijenta, vidi se dosta sličnosti sa UDP klijentom:

```
import socket

PORT = 35789

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Metoda `socket()` se poziva u oba klijenta, s tim što se koristi drugačiji tip soketa. U slučaju UDP-a slučaju je to `SOCK_DGRAM` – datagram soket, a prilikom kreiranja TCP soketa koristimo `SOCK_STREAM` tip soketa, koji je namenjen TCP komunikaciji. Prvi parametar je isti kao i kod UDP-a i označava verziju IP protokola koji se koristi.

U sledećoj liniji se već vide razlike između ova dva klijenta. Naime, pošto pre ostvarivanja međusobne konekcije nije moguć prenos podataka preko TCP-a, potrebno je tu konekciju ostvariti. Ovu inicijalizaciju konekcije vršimo metodom `connect()`:

```
client_socket.connect(('127.0.0.1', PORT))
```

Metoda `connect()` prima za parametar n-torku sa dva elementa:

- adresu na kojoj server očekuje konekciju;
- broj porta preko kojeg će se odvijati konekcija.

Nakon izvršavanja ove konekcije, obavili smo čitavo trostruko usaglašavanje sa serverom i komunikacija može da počne. Kada se implementira TCP server, dobro je kodirati upravljanje grešaka u ovoj tački, jer ako proces usaglašavanja sa serverom ne uspe, na klijentskoj strani će doći do greške `ConnectionRefusedError: [WinError 10061]` – i ovo se razlikuje od implementacije UDP klijenta, gde će program jednostavno stati i čekati da se u nekom trenutku poveže sa serverom.

Pošto TCP održava redosled paketa, moguće je da pošalje prvo jedan pa drugi deo paketa. Slanje poruke od klijenta ka serveru – tačnije najpre pretvaranje poruke u pakete, pa slanje serveru – vrši se pomoću dve metode: `send()` i `sendall()`. Ako koristimo metodu `send()`, moramo imati na umu da se možda neće poslati cela poruka, u zavisnosti od popunjenosti reda za slanje paketa na nivou mrežne kartice. Dobra strana je što metoda `send()` ima za povratnu vrednost broj bajtova koji je zapravo poslat, pa na osnovu toga možemo ručno pratiti koliko još od željene poruke treba poslati. Sa druge strane, metoda `sendall()` ovaj problem rešava umesto nas. Slanje poruke izgleda ovako:

```
client_socket.sendall(bytes('Hello, there, server!', encoding = 'utf8'))
```

Odgovor servera se dobavlja pomoću `recv()` metode, kao i kod UDP klijenta:
`input_s_modified = client_socket.recv(1024)`

Na kraju, soket zatvaramo metodom `close()`: `client_socket.close()`

Primer TCP klijenta je smešten u fajl `tcp_local_client.py` i izgleda ovako:

tcp_local_client.py

```
import socket
PORT = 35780
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
client_socket.connect(('127.0.0.1', PORT))
print ("[CLIENT] Client's has received from server \
its dedicated socket: {}".format(client_socket.getsockname()))
client_socket.sendall(bytes('Hello, there, server!', encoding =
'utf8'))
reply = client_socket.recv(1024)
print ('[CLIENT] Response from the server:
"{}"'.format(reply.decode('utf8'))))
client_socket.close()
```

TCP Server

Kada je reč o poređenju TCP i UDP servera, tu već postoji dosta razlika. Prvenstveno jer je na serverskoj strani zapravo potrebno kreirati dva različita soketa kako bi se TCP server pravilno implementirao.

Kao i kod TCP klijenta, potrebno je definisati promenljivu tipa soket sa tipom `SOCK_STREAM` na verziji 4 internet protokola:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Sam soket ćemo povezati na lokalnu adresu i port našeg uređaja komandom `bind()`:

```
server_socket.bind(('127.0.0.1', PORT))
```

Ovom linijom smo kreirali prvobitni soket koji će služiti za uspostavljanje komunikacije sa klijentom. Važno je napomenuti da se u ovom trenutku, uključujući i tu liniju, još uvek ne zna da li server želi da komunicira preko nadolazeće konekcije. Taj trenutak dolazi nakon korišćenja metode `listen()` – metode kojom osluškujemo nadolazeće konekcije. Kada koristimo ovu metodu, naš dosadašnji soket objekat ne možemo više koristiti za konvencionalno primanje i slanje podataka, već je jedini način da ovaj soket primi nadolazeću komunikaciju korišćenje do sada neupotrebljavane metode `accept()`. Ova metoda je jedinstvena za TCP protokol i vraća novi soket objekat (novi soket objekat i njegovu adresu) – koji ćemo u kodu zvati `conn_socket`, jedinstven samo tom klijentu koji je uspostavio konekciju. Metoda `listen()` prima jedan parametar tipa `int` sa minimalnom vrednošću 1, gde taj broj predstavlja broj konekcija dozvoljenih ovom soketu. Ovaj deo koda izgleda ovako:

```
server_socket.listen(1)

conn_socket, conn_sockname = server_socket.accept()
```

Nakon `accept()` metode završen je proces trostrukog usaglašavanja (three-way handshake) i slanje podataka između klijenta i servera može početi.

Pošto je omogućena klijent–server komunikacija, sledi implementacija potrebne logike, koja je bila i zadatak ovog primera:

```
modified_message = str(len(str(message).split(", ")))

conn_socket.sendall(bytes(modified_message, encoding = 'utf8'))
```

I u realizaciji serverske strane takođe umesto `send()` metode koristimo mnogo pouzdaniju `sendall()` metodu.

Takođe je moguće TCP server implementirati uz pomoć beskonačne `while` petlje, kao što je slučaj i kod implementacije UDP servera; ona se postavlja nakon `listen()` metode.

Ako koristimo `while` petlju, na kraju svake iteracije, potrebno je metodom `close()` zatvoriti soket objekat kreiran samo za potrebe komunikacije za dati klijent (`conn_socket`). Pri izlasku iz petlje, zatvoriti glavni serverski soket (`server_socket`).

Primer implementacije TCP soketa sa korišćenje `while` beskonačne petlje smešten u fajl `tcp_local_server.py` izgleda ovako:

tcp_local_server.py

```
import socket, sys
PORT = 35780
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('127.0.0.1', PORT))
server_socket.listen(1)
while True:
    print ('[TCP_SERVER] Listening at:
    {}'.format(server_socket.getsockname()))
    conn_socket, conn_sockname = server_socket.accept()
    print ('[TCP_SERVER] Connection is accepted from:
    {}'.format(conn_sockname))
    print ('[TCP_SERVER] Socket connects: server_socket: {} and
    conn_socket: {}'.format(
        conn_socket.getsockname(), conn_socket.getpeername()))
    message = conn_socket.recv(1024)
    print ('[TCP_SERVER] The client_socket at {}, originally sent
    "{}"'.format(conn_socket.getpeername(), message.decode('utf8')))
    modified_message = str(len(str(message).split(", ")))
    conn_socket.sendall(bytes(modified_message, encoding = 'utf8'))
    conn_socket.close()
    print ("[TCP_SERVER] Reply sent, closing client's socket.")

server_socket.close()
```

Pitanje

Kojom metodom osluškujemo za nadolazeće konekcije na serverskoj strani?

- send()
- **listen()**
- hear()

Objašnjenje:

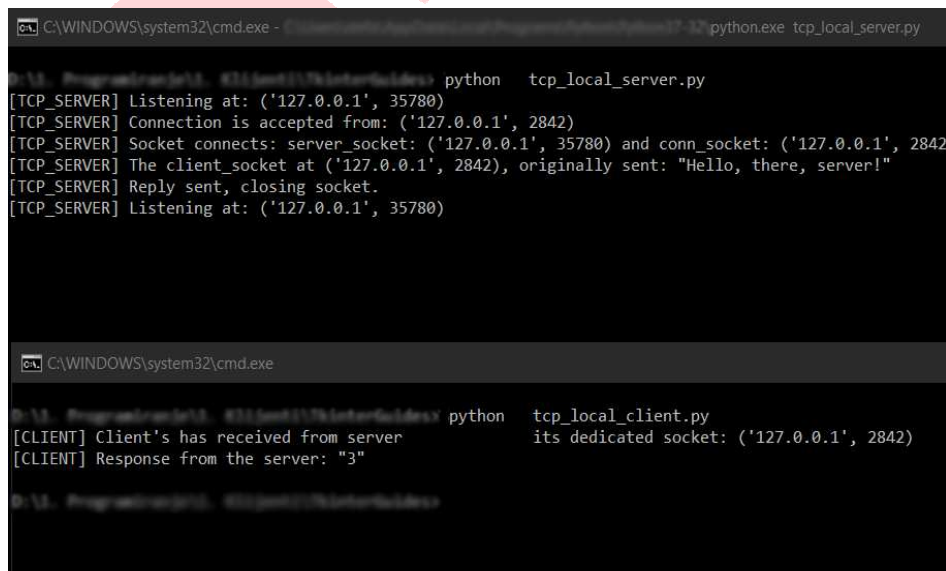
Tačan odgovor je da se nadolazeće konekcije na serverskoj strani osluškuju metodom listen().

Pokretanje i analiza rezultata

Kako bismo, u toku rada programa, pažljivije ispratili adresu i broj soketa u komunikaciji između klijenta i servera, na raspolaganju su nam dve često korišćene metode:

- getsockname – ovom metodom dobivamo tačnu adresu i port datog soketa; može se koristiti i na serverskoj i na klijentskoj strani;
- getpeername – ovom metodom dobivamo na koju je udaljenu adresu i port soket povezan.

Kako bismo proverili funkcionalnost TCP klijenta i servera, moramo pokrenuti oba fajla, što ćemo najlakše učiniti pomoću komandne linije. Dakle, prvo u zasebnom prozoru komandne linije treba pokrenuti tcp_local_server.py, a potom otvoriti novi prozor komandne linije i iz njega pokrenuti fajl tcp_local_client.py.



```
C:\WINDOWS\system32\cmd.exe - C:\Users\user\AppData\Local\Programs\Python\Python37-32\python.exe tcp_local_server.py
D:\16. Programiranje\16.1. Klijent i server\server\python tcp_local_server.py
[TCP_SERVER] Listening at: ('127.0.0.1', 35780)
[TCP_SERVER] Connection is accepted from: ('127.0.0.1', 2842)
[TCP_SERVER] Socket connects: server_socket: ('127.0.0.1', 35780) and conn_socket: ('127.0.0.1', 2842)
[TCP_SERVER] The client_socket at ('127.0.0.1', 2842), originally sent: "Hello, there, server!"
[TCP_SERVER] Reply sent, closing socket.
[TCP_SERVER] Listening at: ('127.0.0.1', 35780)

C:\WINDOWS\system32\cmd.exe
D:\16. Programiranje\16.1. Klijent i server\client\python tcp_local_client.py
[CLIENT] Client's has received from server its dedicated socket: ('127.0.0.1', 2842)
[CLIENT] Response from the server: "3"
```

Slika 5.1. Prikaz pokretanja klijenta i servera kroz komandnu liniju

Nakon što pokrenemo serverski deo, komandni prozor će samo ispisati liniju: `[TCP_SERVER] Listening at: ('127.0.0.1', 35780)` i stati tu. U tom trenutku server je u stanju čekanja i osluškuje na datoj adresi i portu. Pošto pokrenemo klijentski kod, server će ispisati sledeće linije:

- `[TCP_SERVER] Connection is accepted from: ('127.0.0.1', 2842)` – Klijent i server su prošli kroz trostruko usaglašavanje (three-way handshake) i konekcija je ostvarena (metoda `accept`). Kreiran je novi soket, po imenu `conn_socket`, koji će biti zadužen za dalju komunikaciju. Zahtev je stigao od strane klijenta i to sa adrese `127.0.0.1` i porta `2842`.
- `[TCP_SERVER] Socket connects: server_socket: ('127.0.0.1', 35780) and conn_socket: ('127.0.0.1', 2842)` – Na ovoj liniji vidimo razliku između serverskog, glavnog soketa, koji služi za osluškivanje dolazećih konekcija, i soketa koji se kreira kada se komunikacija uspostavi.

Pri uspostavljanju komunikacije, klijent odmah nastavlja ka liniji `sendall()`, preko koje šalje poruku, dok serverski soket dolazi u stanje čekanja na prijem nove poruke koristeći metodu `recv()`. Nakon primanja te poruke ispisuje se linija: `[TCP_SERVER] The client_socket at ('127.0.0.1', 35780), originally sent: "Hello, there, server!"`. Njome ispisujemo samo primljenu poruku i adresu sa koje dolazi (klijentsku adresu i soket).

Dalje se vrši podela stringa koristeći zapetu kao znak za razdvajanje i rezultat se šalje klijentu preko metode `sendall()`, dok će klijent primiti tu poruku metodom `recv()`.

Rezime

- Za implementaciju TCP servera i klijenta se koristi ugrađena Python biblioteka `socket`, koja u sebi ima već ugrađenu funkcionalnost da sama odradi trostruko usaglašavanje.
- Kao tip soketa koristimo `SOCK_STREAM`.
- Za slanje poruka koristimo `sendall()` metodu, kojoj prosleđujemo `bytes` tip objekta.
- Metodu `listen()` koristimo za osluškivanje nadolazećih konekcija na serverskoj strani.
- Metodu `accept()` koristimo na serverskoj strani i ona vraća novi objekat tipa soket i tek sa njim možemo obavljati slanje i primanje poruka. Taj novi soketski objekat je potrebno zatvoriti nakon korišćenja, a pre ponovnog korišćenja `accept()` metode.